

Preliminary Design of a Unified JML Representation and Software Infrastructure*

Robby
SAnToS Laboratory, Kansas State University
robby@cis.ksu.edu

Patrice Chalin
DSRG, Concordia University
chalin@dsrc.org

ABSTRACT

As a Behavioral Interface Specification Language (BISL) for Java, the Java Modeling Language (JML) is tightly coupled to the base language it enhances. Up until Java 1.4, JML kept apace with the evolution of its base language. Java 5 and subsequent revisions have yet to be fully supported by JML tools. Recent efforts such as JML4 have been addressing this issue by providing an Eclipse-based tooling infrastructure. In its current form, JML4 has a fairly steep learning curve for developers wishing to contribute to or extend it. To address this issue and bring JML tools one step closer to a desired plug-in model, we propose a JML Intermediate Representation (JIR) and supporting software infrastructure for JML front-ends and back-ends.

1. INTRODUCTION

The Java Modeling Language (JML) is a design- and code-level assertion-based contract language for Java [18]. It is part of a class of specification languages called Behavioral Interface Specification Languages (BISLs) that are intimately coupled to the underlying programming language they extend [16].

Unfortunately—or fortunately, depending on the point of view—the Java language is evolving at a rapid pace: as an indication of this, we note Sun’s End Of Service Life (EOSL) for Java 1.4, 5 and 6 are 2008, 2009 and 2010, respectively. This has made it difficult for developers of first-generation JML tools to keep their tools up-to-date because they were all responsible for the evolution of their own underlying Java front-end or compiler as well! Recent efforts, such as JML4 [7], have been addressing this issue by providing an industrial-grade software infrastructure for JML tools developed on top of the Eclipse Java Development Tools (JDT)—whose maintenance is in the able hands of the Eclipse Foundation. Unfortunately, in its current form, JML4 has a fairly steep learning curve and maintenance overhead for developers wishing to contribute to it or extend it. This is partly due to the fact that JML4’s core is built atop the internal (non-public) JDT core services and, the commitment of keeping it in-sync with JDT’s bi-weekly builds (including running almost thirty thousand JDT and JML4 test cases); as a first attempt at providing a next-generation platform for JML tools built upon the JDT, this was deemed the most appropriate approach at the time.

*This work was supported in part by the US National Science Foundation (NSF) award CNS-0709169 and CAREER award CCF-0644288, and by the Natural Sciences and Engineering Research Council of Canada.

To address the above issues, we propose to decouple front-ends and back-ends by means of a pragmatic, unified Intermediate Representation (IR) and supporting software infrastructure. Our proposed JML IR (JIR) is based on the Java 5 metadata (annotation) facility: among other things, the Java 5 metadata facility offers a standard, rudimentary but effective means of extending Java’s syntax as well as a standard means of embedding metadata in class files.

After a brief introduction to JML, its classical syntax, and a Java 5 annotation-based syntax (Sec. 2), we proceed to cover the main contributions of our work:

- We enumerate design goals (Sec. 3.1), along with their rationale, that we believe any intermediate JML representation and its supporting software infrastructure should satisfy.
- We propose as an intermediate representation for JML the use of Java 5 annotations with a practical encoding of JML expressions as pure Java expressions (Sec. 3). A key advantage of JIR is that it does not require a dedicated JML processor. This removes a significant engineering and maintenance overhead for the JIR software infrastructure (Sec. 3.3) as well as for its users.
- We describe how JIR can be used as a common intermediate representation for different JML front-ends (e.g., JML2, JML4 [7] and OpenJML [10]) and tooling back-ends (Kiasan, RAC, ESC4, and JMLDoc) including its current implementation in JML4.5 and Kiasan [13] (Sec. 4). We close the paper with an assessment of the manner in which we believe our stated design goals have been met to a large extent (Sec. 5), related work (Sec. 6) and finally, conclusions and future work (Sec. 7).

2. JML AND ITS SYNTAX: CLASSICAL AND NEW

A simple counter class containing JML annotations written using the classical JML syntax is given in Fig. 1. The class gives an example of JML modifiers (`spec_public`), a class invariant as well as method contracts having preconditions (`requires`) and postconditions (`ensures`).

Soon after the release of Java 5, Boysen-Taylor performed a comprehensive study [24] of the alternate ways in which JML could be encoded as Java 5 annotations and came up with a recommended syntax—sometimes referred to as JML5. (Unfortunately, Boysen-Taylor facing the challenge of adding support for Java 5 annotations to a base Java 1.4 compiler, JML5 tool support was prototyped but never officially released.) JML4.5 is an extended version of JML4

which supports a variant of the JML5 syntax¹. A JML4.5 version of Counter is given in Fig. 2. Note how there are Java 5 annotation types for the various JML clauses and that, for example, arguments to the clauses are contained within the Java Strings. JML4.5 is the tool base used to prototype the realization of the intermediate representation for JML proposed in the next section.

3. JML INTERMEDIATE REPRESENTATION (JIR)

3.1 Design Goals and Rationale

An intermediate JML representation is mainly aimed at providing a unified representation for JML that can be easily generated by various JML front-ends and consumed by different JML back-ends. Below are design goals and corresponding rationale that we believe any intermediate representation and its supporting infrastructure should satisfy to some extent:

- D1.** *Low barrier of (re-)entry:* To help ease adoption, JML tool developers should be able to learn JIR without significant investments, and to use its tool support without a significant learning (and maintenance) overhead.
- D2.** *Comprehensive:* All of JML constructs should be representable in JIR.
- D3.** *Extensible:* JIR should be easy to extend to handle custom JML constructs for experimentation with new language features.
- D4.** *Implementation-independent:* JIR should not be tied into a particular JML front-end, nor should it be biased toward a particular analysis technique (e.g., static or runtime assertion checking) or back-end.
- D5.** *Robust tool-support for processing:* JIR tool support should be based on stable and robust software infrastructures. In order not to compromise the robustness of the underlying infrastructures, the JIR code-base should be small, thus easily maintained. Creating an alternative implementation of JIR should take little to modest effort, hence ensuring JIR’s implementation-independence.
- D6.** *Can be tightly integrated in various IDEs:* IDEs play an important role in the development of modern software. To help ensure adoption, it should be possible to tightly integrate JIR in popular Java IDEs.
- D7.** *Can easily be constructed by hand:* While JIR is targeted for automatic processing, it should also be relatively easy to construct JIR specifications manually. This makes writing test cases easier and allows tool developers working on JML extensions to prototype and experiment with their extension even without a supporting JML front-end.

3.2 JIR Definition

While developing JIR, we used the previously stated design goals as a guide; for conflicting goals, we opted for function and ease in engineering/maintenance over form (i.e., JIR is not intended as JML input syntax for end-users).

¹The only difference between JML5 and JML4.5 syntax currently worth highlighting is the use, within annotation element strings, of ‘\$’ instead of ‘#’ to replace the slash character traditionally used in JML: e.g., “\$result” vs. “#result” and “\result”.

Thus, we settled on a pure Java representation of JML using Java 5 annotations. (For simplicity we assume JIR will only be used/applied to well-formed Java code.) In contrast to JML4.5’s input syntax however, JML expressions are encoded in JIR as pure Java expressions which also embed, e.g., typing and “symbol table” information as will be explained shortly. A JIR encoding of Fig. 2 is illustrated in Fig. 3. Notice how, e.g., JML5 annotations are represented by similarly named JIR annotations. In contrast, one can see a marked difference between the original pre/post-condition predicates and their JIR encoding.

In a sense, the JIR encoding of JML expressions is a source representation of a mixture of source and bytecode level information. More specifically: (a) all JIR expressions are well-formed (hence also well-typed) Java expressions; (b) most “symbol table” information is explicitly represented; (c) source-level traceability information for symbols is encoded; (d) type information for local and quantified variables is also explicitly encoded; (e) JML constructs such as \result and \old are encoded as *method invocations*; and (f) regular Java expressions in JML are represented as Java expressions in JIR.

For example, assuming we have an int local variable x, \old(x) is represented as `JIR.old(JIR.local("x", int.class, 10, 4, 1))`², where 10 and 4 are the source line and column of where x is declared, 1 is the local slot of x at the bytecode level, and JIR is a JIR class with helper methods such `old` and `local` to represent JML constructs as method invocations and declared as:

```
public static native <T> T old(T o);
public static native <T> T local(String id,
                                Class<T> cls, int l, int c, int s);
```

In essence, the Java method invocation syntax is used as a *representation* for JML constructs. The Java encoding is not designed to be executable as is; what matters is its *interpretation* by JML tools back-ends that consume JIR expressions. In a sense, Java method invocation syntax is being used in JIR as (typed) S-expressions [22].

We describe how JML clauses and expressions are represented in more detail below; due to space limitations, we chose some representative JML constructs.

Type and method specifications: As alluded above, type and method specifications are represented using Java annotations. JIR uses JML5’s annotation schema—see Fig. 3. As can be observed, JML-specific expressions are encoded in JIR similar to what has been discussed above. For example, this is represented as `JIR.receiver(Counter.class); JIR.local("this", Counter.class, 0, 0, 0)` can be used, but the former is a more compact representation without loss of information.

Quantified expressions: JML quantifications are also encoded as method invocations. One key difference as compared to other JML constructs is that quantified expressions declare new variables that are later used in their body. Quantified variables are encoded like local variable references. For example, `(\forall int i; ... i ...)` is encoded as `JIR.forall(int.class, "i", ... JIR.qvar("i", int.class) ...)`.

Model specifications: There are three categories of JML

²Generic types can be preserved using the Java type casting syntax; auto-boxing/unboxing are used to resolve issues with primitive/non-primitive scalars.

```

// Counters that count up to MAX and then wrap back to 0.
public class Counter {
    public final static int MAX = 100;

    /*@spec_public*/ private int count;
    //@ public invariant 0 <= count && count <= MAX;
    // ... constructor not shown.

    //@ ensures \result == count;
    /*@pure*/ public int getCount() {
        return count;
    }

    /*@ requires count < MAX;
    @ ensures count == \old(count) + 1;
    @ also
    @ requires count == MAX;
    @ ensures count == 0;
    @*/
    public void inc() {
        count = count < MAX ? count + 1 : 0;
    }
}

```

Figure 1: Classical JML annotations in stylized Java comments (excerpts)

```

import org.jmlspecs.annotation.*;
import static org.jmlspecs.JML.*; // defines $result, etc.

public class Counter {
    ... // elided
    @SpecPublic private int count;
    ...
    @Ensures("$result == count")
    @Pure public int getCount() { ... }

    @Also({@SpecCase(requires = "count < MAX",
                    ensures = "count == $old(count) + 1"),
           @SpecCase(requires = "count == MAX",
                    ensures = "count == 0") })
    public void inc() { ... }
}

```

Figure 2: Sample JML4.5 syntax using Java 5 annotations (excerpts)

```

import org.jmlspecs.jir.annotation.*;

public class Counter {
    ... // some details elided
    @SpecPublic private int count;
    ...
    @Ensures("JIR.result(int.class) == JIR.receiver(Counter.class).count")
    @Pure public int getCount() { ... }

    @Also({@SpecCase(
        requires = "JIR.receiver(Counter.class).count < Counter.MAX",
        ensures = "... JIR.old(JIR.receiver(Counter.class).count)..."),
        ... })
    public void inc() { ... }
}

```

Figure 3: The Counter class example in JIR (excerpts)

model specifications: (1) model types, (2) model fields, and (3) model methods³. Each of these is represented in JIR by real Java classes, fields, and methods (marked with `@Model`), respectively. By making these specification elements explicit in JIR, support for model specifications become simpler because symbols associated with model attributes can be resolved as for regular attributes. (A simple checker can be developed to notify users when regular code refer to JIR model elements.)

Also, at first thought, one might believe that this approach to representing, e.g., model methods by real methods could affect analysis back-end tools. For example, a program that uses Java reflection to iterate over the methods declared in a class now has more methods to iterate over. Actually, independent of JIR, this is already the case. That is, Java 5 compilers sometimes add extra methods, called *bridge* methods, to help deal with issues that arise due to type erasure in bytecode. Since analysis tools already have to take into account bridge methods, it will be little extra work to filter out `@Model` methods.

Inline specifications: Java 5 annotations are currently limited when it comes to their use inside static blocks or constructor and method bodies; i.e., in such cases, annotations can only be applied to (local or catch) variable declarations. This makes it difficult to represent JML inline specifications such as `assert`. To address this, we adopt the approach taken in the Bandera Specification Language (BSL) [12] which leverages Java labels to indicate program points inside methods. For example, we use the method annotation: `@Maintaining("L", 14, 20, ...)` to represent the following loop invariant at label L (assuming L is at source line 14 and bytecode offset 20):

```
//@ maintaining ...;
L: for (int i = 0; ...) { ... }
```

3.3 JIR Software Infrastructure

In order to alleviate the overhead associated with using JIR, we propose a lightweight JIR software support infrastructure, consisting of a shared library and two major standalone tools, which makes use of the stable API of robust third party components such as the ASM bytecode engineering framework [25] and the Eclipse Java Domain Object Model (DOM). These two major tools are: (1) JIR Specification Embedder, and (2) JIR Specification Extractor. Both components can work with source code and class files, and they communicate with JML tools using the same input/output format.

Specification Embedder: The embedder is the component that interfaces with JML front-ends. To use the JIR infrastructure, a JML front-end transforms JML specifications into JIR (as will be described in the input/output section below). The JIR annotations are then embedded in Java class files using the ASM bytecode engineering framework that provides an API for transforming Java class files. ASM is a mature open source tool (which happens to be shipped with Eclipse 3.5). An alternative implementation could be done using the BCEL library [26]. In addition to transforming Java class files, the embedder also provides an API to build Java ASTs with embedded JIR annotations (which, e.g., can be pretty printed for debugging purposes).

³Ghost fields are essentially treated like model fields.

The embedder makes use of the Eclipse Java Development Tools (JDT) public API and Java AST DOM, both of which are stable, well-maintained and have well-defined APIs.

Specification Extractor: The extractor interfaces with JML back-ends, and it provides an API to retrieve JIR specifications from Java classes using ASM. In addition to delivering JML expressions as strings, the JIR extractor provides an API for parsing and building Java Abstract Syntax Trees (ASTs). Moreover, type analysis can also be performed to build type bindings (though some back-ends, such as Kiasan, do not need type information for expressions). Alternatively, JIR annotations can be extracted from source code. Similar to the embedder, AST manipulation (and type binding information) is done using the JDT public API.

Input/Output (I/O) Format: Both front/back-ends communicate in a similar way with the JIR infrastructure, i.e., either through (a) an in-memory representation, or (b) a XML representation. The in-memory representation can be used for front/back-ends that can access its Java API, and the XML representation can be used for tools that are not implemented in Java, or offline operation. The I/O format for both representations are finite maps (\rightarrow) with the following schema:

- $class\ name \rightarrow field\ name \rightarrow annotation\ name \rightarrow element\ name \rightarrow value$
- $class\ name \rightarrow method\ signature \rightarrow annotation\ name \rightarrow element\ name \rightarrow value$
- $class\ name \rightarrow method\ signature \rightarrow parameter\ index \rightarrow annotation\ name \rightarrow element\ name \rightarrow value$

With the exception of the method *parameter index* and *value*, everything else is a string. *class name* (and *annotation name*) are fully qualified names of a class (or annotation), *field name* is a name of a class field, *element name* is a name of an annotation element, and *method signature* is the unique signature of a method. *value* can be of type Byte, Boolean, Character, Short, Integer, Long, Float, Double, String, or `java.lang.Class`. In addition, a value can be a nested annotation or an array of values (of the same type). As mentioned above, JIR expressions encoded as String can be parsed to produce `org.eclipse.jdt.core.dom` Java AST classes.

4. INTEGRATING JML FRONT-ENDS AND BACK-ENDS

In this section, we describe how some JML tools have been adapted, and how we foresee others can be adapted, to leverage JIR and its software infrastructure. Both front-ends and back-ends are described. For back-ends, we assume that each tool at least has the capability for processing regular Java programs, because the JIR infrastructure provides only an API for processing JML specifications in JIR.

Front-ends: JIR can be used by JML front-ends in two ways: (1) as an interchange format for connecting with JIR-enabled JML back-ends, or (2) as an internal IR for the front-end itself. Option (1) can be used for front-ends that are not actively updated with respect to recent Java language enhancements such as JML2 (i.e., the Common JML Tools [5]). Coupled with a tool that translates JIR back (i.e., pretty printing) to other JML input syntaxes, we then have a JML converter from one input syntax to another; this is useful for salvaging existing specifications of, for example, Java libraries.

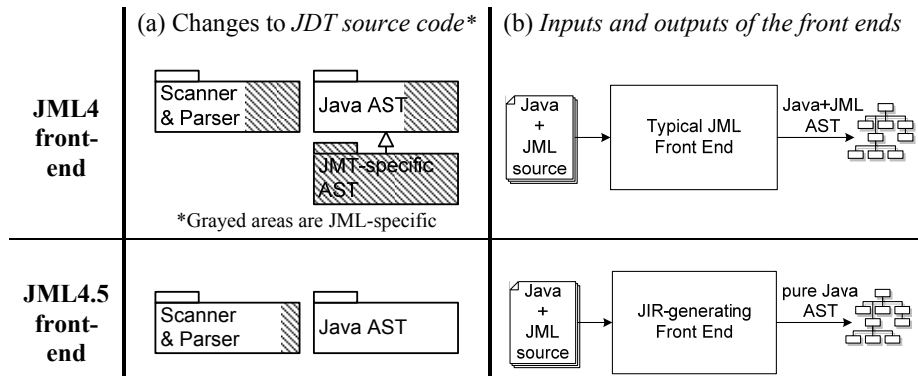


Figure 4: Comparison of Typical vs. JIR-generating JML Front-ends

Option (2) can be adopted to help reduce the engineering and maintenance overhead of JML front-ends. The first row of Fig. 4 illustrates how JML front-ends processing classical JML syntax—such as Common JML Tools, JML4, and OpenJML—are usually implemented. That is, they usually use an underlying compiler framework and enrich its parser and AST in support of JML. Of course, this tightly couples the front-end to the base compiler framework. If the compiler framework is outdated, then it becomes a challenge to maintain the front-end (e.g., JML2 that uses MultiJava [9] which is no longer maintained). If the compiler framework is up-to-date, but imposes a constant engineering and maintenance overhead then most JML tool developers are reluctant to invest in and contribute to it (e.g., JML4).

The second row of Fig. 4 illustrates how a JML front-end, in this case JML4.5, has been adapted to use JIR as its internal representation. JML4.5 still modifies an underlying compiler framework’s parser, but it uses Java annotations for representing JML specifications, which can then be resolved to produce JIR specifications. This approach reduces the amount of coupling to the underlying framework, and hence is more manageable to maintain. In addition, back-end tool developers are completely decoupled from the front-end.

Back-ends: To illustrate how JML back-ends can benefit from the use of JIR, we discuss the cases of 3 analysis tools that use different analysis techniques (i.e., Kiasan, JML RAC, and ESC/Java) as well as a non-analysis tool (i.e., JMLDoc).

Kiasan: Kiasan is a static analyzer similar to ESC/Java that is able to analyze deep semantical properties of complex heap structures and to generate counter-examples or models (e.g., test cases). Its first generation, Bogor/Kiasan, used Java annotations for expressing JML type and method specifications similar to JML5, but it uses pure Java expressions for representing JML expressions [13]. Special JML expressions such as `\old` were represented as Java method invocations similar to JIR, except that they were strictly processed at the source-level, untyped, and unresolved. That is, before analysis, an annotation processor would transform and embed JML specifications into the source code (e.g., embedding `requires` clause predicates at method entry points), which would then be compiled using a regular Java compiler.

The second (current) generation of Kiasan, Sireum/Kiasan, uses an extension of JML4 called JMLK to process JML specifications. During execution, when Kiasan substitutes a method implementation with its contract, Kiasan

dynamically invokes JMLK to compute the *effective* method contract (e.g., conjoining invariants with method pre/post-conditions) and then transforms it into an executable form amenable for Kiasan’s analysis. To compute effective method contracts, JMLK traverses a JML4 AST and generates Kiasan’s contract classes; the classes are then compiled to class files which Kiasan uses for its analysis. While this approach works fine, JMLK is tightly coupled with the internal JDT AST used in JML4; this is undesirable as it depends on an unstable API.

Using JIR alleviates the above issue. That is, JMLK now calls the JIR API instead of the JML4 API to compute effective method contracts. Since the JIR I/O format presented in Sec. 3.3 is relatively simple and the JIR infrastructure provides an API to build stable ASTs for JML expressions, JMLK works as it did before when using JIR. In addition, the translation schema for Kiasan executable contract classes do not need to be changed.

JML Runtime Assertion Checker (RAC): Similar to JML front-ends, JML RAC is usually implemented as an extension of an underlying compiler framework (e.g., JML2 RAC [8], JML4). Thus, the same coupling issue is present and JIR can also help in this context to reduce coupling. That is, instead of being tightly integrated to the internal processing of a base compiler, RAC can be implemented via bytecode instrumentation. This approach is implemented by Modern Jass [21] (which actually accepts annotations in the format of JML5) and is currently being prototyped in JML4.5.

At the time that various platforms were being explored as candidates for a next generation tooling framework for JML, Gary Leavens and others in the JML community proposed implementing JML RAC using aspects and, in the context of Eclipse, AspectJ. In fact, such a use of AspectJ has been implemented (using the JML2 front-end) by Rebêlo *et al* [20]. Unfortunately, due to JML4’s tight coupling to the JDT compiler, this possibility had been ruled out (because AspectJ is itself a variant of the JDT compiler). Now, with the looser coupling of JML4.5 to the JDT, it becomes feasible to explore this avenue.

ESC/Java: Similar to RAC, ESC/Java frameworks (i.e., ESC/Java [15], ESC/Java2 [11], and ESC4 [17]) are tightly integrated with a JML front-end or compiler framework; thus, it can similarly benefit from JIR to decouple its implementation. For example, ESC4 uses the JML4 AST (which depends on JDT’s internal API) for both processing JML specifications and Java programs. When adapted to JIR, ESC4

can depend on JIR API to process JML specifications and JDT public and stable API for processing Java programs.

JMLDoc: JMLDoc is a JML tool that generates HTML documentation in the format of Javadoc [27] that includes JML specifications in addition to the regular Javadoc documentation elements [19]. JMLDoc is implemented as a Javadoc’s doclet, and it uses the MultiJava [9] compiler framework to process JML specifications. When adapted to use JIR, there are two alternative approaches: (1) use the doclet API to process JIR Java annotations, or (2) use the JIR I/O format. While different, the two APIs are similar since the Java annotation form is standard. Regardless of the choice, JMLDoc can still use JIR AST builder API for JML expressions to replace the outdated MultiJava implementation.

5. ASSESSMENT

To a large extent, JIR and its supporting infrastructure satisfy the design goals established in Sec. 3.1. The fact that JIR is pure Java presents a low barrier of entry (**D1**), because JML users do not need to learn a new format. Moreover, the representation is front-end/back-end independent (**D4**). In addition, existing and robust Java compiler (e.g., OpenJDK, Eclipse JDT) and bytecode engineering frameworks (e.g., ASM [25], BCEL [26]) can be used as is without modification; hence, JIR supporting infrastructure can be implemented within a small core codebase (**D5**). This also means that the JIR infrastructure can be tightly-integrated in popular Java IDEs such as Eclipse or NetBeans (**D6**). Since JIR uses Java annotations and the method invocation syntax whose schema can be easily modified/enhanced by tool developers, JIR can represent any JML construct and can be easily extended (**D2** and **D3**). In addition, JIR is Java-based at a mixed source/bytecode level that is more amenable for manual construction/modification unlike pure S-expressions or XML (**D7**).

Perhaps one of the main weaknesses of JIR is that its encoding of JML expressions is a verbose (e.g., symbol table embedding) and non-traditional use of Java syntax. It is non-traditional in that the resulting encoding essentially represents a fully resolved AST which, if it contains encoding of JML constructs, has no (precise) meaning until it is interpreted by a back-end tool. Our choice of encoding was a difficult choice that we made in order to liberate ourselves from the engineering and maintenance burden of having a dedicated JML processor or being tightly coupled with an unstable (internal) compiler API. However, since JIR is designed mainly for automatic processing, a representation is just that, a representation. Considering JML’s tool history, its current state, the scale of the problem with the ever expanding Java language features, and the number of active contributors/benefactors, we believe this is a worthwhile compromise. In the end, what really matter is not the form of the JML intermediate representation, but the impact that JML and its tools make on formal method research and software reliability in general.

6. RELATED WORK

We have discussed similarities and differences with some existing approaches throughout the paper. Due to space limitations, we only focus on a few other closely related approaches in this section. One such approach is the Bytecode Modeling Language (BML), a bytecode JML represen-

tation [6]. Similar to BML, JIR uses a mixed source/bytecode levels for JML expressions; however, JIR uses Java annotations and pure Java expressions that do not require a dedicated processor. The two are complementary, and JIR can easily be translated to BML for tools using BML.

The Microsoft Code Contract project goes beyond our modest goal of providing a means of representing contracts for Java in a tool-independent way. At the heart of the Code Contract approach is a library of static methods and conventions for their use. For example, the postcondition of a user method increasing the value of the field x could be encoded by placing, at the start of the method body, the call to `Contract.Ensures(x > Contract.Old(x))`.

JIR and Code Contract are similar in that they both use method invocations to represent special contract expressions (e.g., `JIR.old` and `Contract.Old`). A similar encoding has been used in other approaches such as in the Bogor software model checking framework [23] for providing language extension mechanisms to model domain-specific abstractions and optimizations (e.g., [14, 1, 3]). Kiasan [13] adopts the same approach to implement custom extensions for interpreting JML-specific expressions [28]. Another example is the Java PathFinder (JPF) [4] that uses native methods as place holders to facilitate extensions.

In contrast to JIR, Code Contract specification elements, such as method pre/post-conditions, are written as actual code within the method body. Hence, no special front-end is needed to process them. Since they are written as (plain) code, IDE features are available for use over the contract elements. This can be done because .NET compilers feature generic conditional compilation that can be used to strip out contract code, which alleviates some developers’ concern of polluting deployed code with contracts.

Finally, we note that Code Contract tools (a static analyzer and a runtime assertion checker) operate on the bytecode level, while JIR is designed to be used both at the source level and at the bytecode level. We are able to leverage mature Java frameworks such as Eclipse JDT and ASM for specification processing for JIR while corresponding frameworks for .NET are still in active development [2].

7. CONCLUSION AND FUTURE WORK

This paper proposes a JML intermediate representation and supporting infrastructure that can serve to unify JML front-ends (e.g., JML2, JML4, and OpenJML) and back-ends (e.g., Kiasan, JML RAC, ESC). The infrastructure enables the development of a specification, programming, and verification environment where all JML tools can be tightly integrated, but whose implementations are decoupled with each other; this is what we are actively pursuing to achieve.

We have prototyped most of the JIR infrastructure and assessed its feasibility. Along with the infrastructure, we plan to develop a catalog of JIR expression schema for JML-specific constructs that can serve as a reference for tool developers. We continue to experiment with and assess JIR and its infrastructure, planning a public release soon.

We hope that JIR will foster formal method research leveraging extensive work in JML by helping reduce the significant engineering overhead of its tool-base development.

8. REFERENCES

- [1] Luciano Baresi, Carlo Ghezzi, and Luca Mottola. On accurate automatic verification of publish-subscribe

- architectures. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 199–208, Washington, DC, USA, 2007. IEEE Computer Society.
- [2] Michael Barnett. Personal communication, 2009.
 - [3] Domenico Bianculli, Carlo Ghezzi, and Paola Spoletini. A model checking approach to verify BPEL4WS workflows. In *IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2007, 19-20 June 2007, Newport Beach, California, USA*, pages 13–20, 2007.
 - [4] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, Jul 2000.
 - [5] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.
 - [6] Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2007.
 - [7] Patrice Chalin, Perry R. James, and George Karabotsos. JML4: Towards an industrial grade IVE for Java and next generation research platform for JML. In *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 70–83. Springer, 2008.
 - [8] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Master’s thesis, Iowa State University, 2003.
 - [9] Curtis Clifton, Todd D. Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):517–575, 2006.
 - [10] David R. Cok. OpenJML project. <http://jmlspecs.sourceforge.net/viewvc/jmlspecs/OpenJML>.
 - [11] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 108–128, 2004.
 - [12] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 2002.
 - [13] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k -bounded symbolic execution for checking strong heap properties of open systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 157–166, 2006. (Experiment programs and data are available at <http://bogor.projects.cis.ksu.edu/kiasan/kunit/laziersharp/>).
 - [14] Matthew B. Dwyer, Robby, Oksana Tkachuk, and Willem Visser. Analyzing interaction orderings with model checking. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pages 154–163. IEEE Computer Society, 2004.
 - [15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 2002.
 - [16] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, School of EECS, Mar 2009.
 - [17] Perry R. James and Patrice Chalin. Extended static checking in JML4: Benefits of multiple-prover support. In *ACM Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT), March, 2009, Hawaii, USA*, 2009.
 - [18] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, October 1998.
 - [19] Arun David Raghavan. Design of a JML documentation generator. Master’s thesis, Iowa State University, 2000.
 - [20] Henrique Rebêlo, Ricardo Lima, Márcie Cornélio, and Sérgio Soares. A JML compiler based on AspectJ. In *Proceedings of the 11th International Conference on Software Testing, Verification, and Validation (ICST)*, pages 541–544. IEEE, 2008.
 - [21] Johannes Rieken. Design by contract for Java – revised. Master’s thesis, University of Oldenburg, 2007.
 - [22] Ronald L. Rivest. S-expression internet-draft. <http://people.csail.mit.edu/rivest/Sexp.txt>, 1997.
 - [23] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 267–276, 2003.
 - [24] Kristina B. Taylor. A specification language design for the Java Modeling Language (JML) using Java 5 annotations. Master’s thesis, Iowa State University, 2008.
 - [25] ASM website. <http://asm.objectweb.org/>.
 - [26] BCEL website. <http://jakarta.apache.org/bcel/>.
 - [27] JavaDoc website. <http://java.sun.com/j2se/javadoc/>.
 - [28] Sireum website. <http://www.sireum.org>.

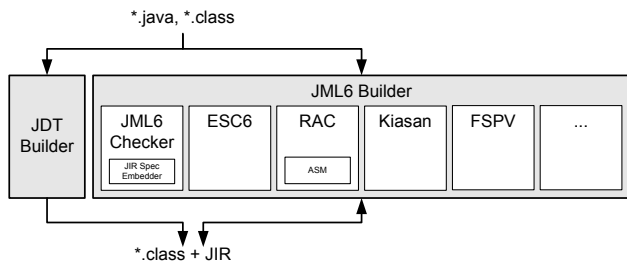


Figure 5: JML6: JML support via a separate builder

APPENDIX

A. JML4.5 AND JML6

As was explained in the main Sec. 2, JML4.5 is a version of JML4 that makes use of Java 5 annotations both as input syntax and intermediate representation. In particular, a variant of JML5 syntax is accepted as input and JIR is used as output format.

JML4.5 is being used as an “incubator” for techniques which we plan to incorporate into JML6, a clean Eclipse plug-in supporting JML checking, RAC, ESC, full static program verification (FSPV), Kiasan as well as other JML-specific “plug-ins”. The objective is to realize JML6 as an Eclipse Builder that will be able to process, *independently* from the JDT, JML annotated Java sources files, altering *.class files to, e.g., insert JIR, ESC status information and/or RAC instrumentation code (Fig. 5). This is in contrast to the manner in which JML4 and JML4.5 are currently implemented, i.e., as “invasive” modifications of the JDT itself. We describe next the ways in which we have improved the design of JML4.5 as compared to JML4.

Some of the techniques that we have begun exploring in JML4.5 for the purpose of decoupling it from the internals of the JDT core include:

- (A) Reducing our footprint on the Scanner and Parser components.
- (B) Adopting JIR.
- (C) Switching to the public API JDT Java DOM

(A): As was briefly explained in Sec. 4, **reducing the JML4.5 footprint on the Scanner and Parser components** is achieved in part by switching to the use of JML5 as an input syntax. Java 5 annotations are used as a means of structuring the JML input syntax—e.g., roughly by having one annotation per JML clause or clause group. What remains to be dealt with in the parser, is essentially the JML extensions to Java expressions. In JML4.5, we can further reduce the need to change the JDT Scanner and Parser by eliminating JML keywords that start with slashes such as `\result`, `\fresh()` and `\old()`. The original use of slashes in these JML keywords was to ensure that they would appear in their own namespaces, hence avoiding any possibility of a conflict with Java code that used similarly named class attributes or parameters. Of course, Java’s main mechanism for avoiding name clashes is packages. Hence, our solution has been to replace the slash prefix character by a ‘\$’ character, thus rendering these new lexemes valid Java identifiers that can naturally be processed by a standard Java scanner. The use of the ‘\$’ character already reduces the likelihood of name clashes with user code. In the unlikely event that a name clash does occur, then fully qual-

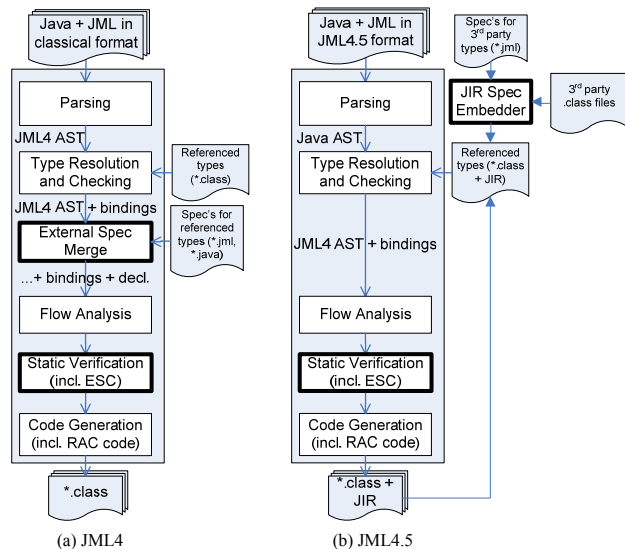


Figure 6: JML4 and JML4.5 compilation phases

ified names can be used for these JML “keywords”—which are actually declared as ordinary Java fields and methods in the class `org.jmlspecs.JML`. This approach allows us to eliminate the special Parser grammar rules for the previously slash-prefixed constructs. Hence, `$old(this.f)` is now parsed as an ordinary method call. We are currently exploring other mechanisms by which some of the other specialized syntax of JML could be similarly rendered more Java-parser friendly. The end result, as illustrated in Fig. 4(a), is significantly reduced coupling and invasive changes to the JDT Scanner and Parser.

(B): Another key decoupling technique is the **adoption of JIR**. Fig. 6 illustrates the differences between the compilation phases as they occur in JML4 and JML4.5. The processing phases in bold represent JML-specific phases. On the left side of the figure we see the JML4 phases. In summary, JML annotated Java source files are parsed and represented internally using the elements of an extended (internal) JDT AST hierarchy. The extensions add AST elements for JML specific constructs. Type resolution and checking has as a side-effect to attach bindings to some of the AST nodes (a binding can be thought of as a semantic object representing an AST element; there are, e.g., type, field, variable and method bindings, among others). This is followed by a JML-specific phase called External Specification Merging (ESM) which serves to look-up the JML source files (*.jml or *.java) for type bindings that were loaded from binary (*.class) files, and attaching the JML AST nodes from the corresponding JML source files. (This is necessary, because binary bindings are void of JML specification information.) Note that, generally speaking, specification merging is only necessary for types referenced by the compilation unit being processed. Beyond this point, the AST and bindings are considered fully resolved and “loaded” with their JML specifications. This information then makes its way through Flow Analysis before being passed on to JML4’s Static Verification components and finally the code generation phase.

Adoption of JML5 syntax and JIR in JML4.5 allows us to

simplify the JML4 compiler phase pipeline. For one thing, Parsing is considerably simplified, thanks to the use of a Java 5 annotation-based input syntax. The output from the Parser in JML4.5 is a Java AST (without JML elements). The JML elements for the compilation unit being processed are contained within Java 5 annotations. These are processed just at the start of the type resolution phase. So that we can continue to use the current JML4 back-end components (which have yet to be adapted to accept JIR), we currently translate JML5 into both JML4 AST nodes and JIR. The JIR of the current compilation unit, being regular Java 5 annotations (with at least CLASS retention), are embedded in the class file by the JDT's standard processing of annotations. The remaining tail-end phases of the compiler pipeline are currently the same as in JML4. JML4.5 does away with the External Specification Merging phase because all JML specification elements are available in the form of JIR attached to binary type bindings. Again, JIR, being stored as Java 5 annotations, are loaded by the JDT during the creation of the binary type bindings. The JIR Specification Embedder is used to attach JIR to "3rd party" classes and libraries for which JML API specifications are available.

(C): Our third decoupling technique was hinted at in Fig. 4(a), namely, a move towards the **adoption of the public JDT Java DOM AST classes**. Being part of the Eclipse public API, the Java DOM is quite stable, hence making it a better target for extension than the internal AST classes. Another advantage that we gain is that these API classes are considerably more fully documented than the internal classes (for which documentation is scarce).

Certain issues with respect to the input syntax of JML4.5 are still being investigated (e.g. quantifiers, and the pragmatics of multi-line JML specifications being difficult to enter in Java 5 annotations). Nonetheless, we are hopeful that the decoupling techniques being experimented with in the context of JML4.5 will help us make better design decisions for JML6 so that, in turn, it may become an accessible and extensible base suitable for use by all JML tool developers.